IEICE Proceeding Series

Memory Reduced Implementation of Error-Free Transformation of Matrix Multiplication and its Performance

Katsuhisa Ozaki, Takeshi Ogita, Shin'ichi Oishi

Vol. 1 pp. 877-880
Publication Date: 2014/03/17
Online ISSN: 2188-5079

2012 International Symposium on Nonlinear Theory and its Applications
NOLTA2012, Palma, Majorca, Spain, October 22-26, 2012

Memory Reduced Implementation of Error-Free Transformation of Matrix Multiplication and its Performance

Katsuhisa Ozaki[1], Takeshi Ogita[2] and Shin'ichi Oishi[3]

1) Department of Mathematical Sciences, Shibaura Institute of Technology
307 Fukasaku, Minuma-ku, Saitama-shi, Saitama 337-8570, Japan
2) Division of Mathematical Sciences, Tokyo Woman's Christian University
2-6-1 Zempukuji, Suginami-ku, Tokyo 167-8585, Japan
3) Faculty of Science and Engineering, Waseda University
3-4-1 Okubo, Shinjyuku-ku, Tokyo 169-8555, Japan
Email: ozaki@sic.shibaura-it.ac.jp, ogita@lab.twcu.ac.jp, oishi@waseda.jp

Abstract—This paper is concerned with accurate numerical algorithms for matrix multiplication. Recently, error-free transformation for matrix multiplication is developed by the authors. It is shown that the transformation is not only useful for accurate numerical computations but also suitable for high performance computing. However, the algorithm requires much amount of working memory. In this paper, how to overcome this drawback is discussed without significant slowdown of the computational performance.

1. Introduction

In this paper, we discuss a fast and accurate algorithm for matrix multiplication by floating-point arithmetic. Floating-point arithmetic is widely used in scientific computing. However, since a floating-point number has finite information, rounding error may occur in each arithmetic operation. Then, we may obtain an inaccurate result in the worst case due to accumulation of rounding errors. A way to overcome this problem may be to use multi-precision libraries, for example GMP, MPFR [6], exflib [7] and so forth. For matrix multiplication, an accurate algorithm for dot product [2] is also useful.

Recently, we proposed an error-free transformation for matrix multiplication [5]. It transforms a matrix multiplication into an unevaluated sum of several floating-point matrices. After this transformation, we can apply accurate summation algorithms, for example, [2, 3, 4] and so forth. It is shown that the transformation is not only useful for accurate numerical computations but also suitable for high performance computing. However, the algorithm requires much an amount of working space since a matrix is divided into several matrices. In this paper, we propose blockwise computations for [5] to overcome this drawback.

2. Previous Work

In this section, we introduce our previous work [5]. Let $\mathbb{F}$ be a set of floating-point numbers as defined by the IEEE 754-2008 standard [1]. Let $\mathbf{u}$ be the relative rounding error unit, for example, $\mathbf{u} = 2^{-53}$ for binary64. $\mathrm{fl}(\cdots)$ means a result of floating-point arithmetic, whose rounding mode is rounding to nearest. For $x, y \in \mathbb{F}^n$, an inequality $x < y$ shows $x_i < y_i$ for $1 \leq i \leq n$. $|x|$ shows $(|x_1|, |x_2|, \ldots, |x_n|)^T$. These notations are similarly extended to matrices, e.g. $A < B$ and $|A|$ for $A, B \in \mathbb{F}^{n\times n}$. Assume that neither overflow and underflow occur in $\mathrm{fl}(\cdots)$.

Let $A = (a_{ij}) \in \mathbb{F}^{m\times n}$, $B = (b_{ij}) \in \mathbb{F}^{n\times p}$ and

$$\beta = \left\lceil \frac{\log_2 n - \log_2 \mathbf{u}}{2} \right\rceil. \tag{1}$$

Two vectors $\sigma \in \mathbb{F}^m$ and $\tau \in \mathbb{F}^p$ are defined by

$$\sigma_i^{(1)} = 2^\beta \cdot 2^{v_i^{(1)}}, \quad \tau_j^{(1)} = 2^\beta \cdot 2^{w_j^{(1)}}$$

where $v^{(1)} \in \mathbb{F}^m$ and $w^{(1)} \in \mathbb{F}^p$ are

$$v_i^{(1)} = \lceil \log_2 \max_{1\leq j\leq n} |a_{ij}| \rceil, \quad w_j^{(1)} = \lceil \log_2 \max_{1\leq i\leq n} |b_{ij}| \rceil. \tag{2}$$

For $e = (1, 1, \ldots, 1)^T$, we compute

$$\begin{aligned} A^{(1)} &= \mathrm{fl}\left((A + \sigma^{(1)} \cdot e^T) - \sigma^{(1)} \cdot e^T\right), \\ \underline{A}^{(2)} &= \mathrm{fl}\left(A - A^{(1)}\right), \\ B^{(1)} &= \mathrm{fl}\left((B + e \cdot (\tau^{(1)})^T) - e \cdot (\tau^{(1)})^T\right), \\ \underline{B}^{(2)} &= \mathrm{fl}\left(B - B^{(1)}\right). \end{aligned} \tag{3}$$

Then

$$A = A^{(1)} + \underline{A}^{(2)}, \quad B = B^{(1)} + \underline{B}^{(2)}.$$

Next, we define $\sigma^{(2)}$ and $\tau^{(2)}$ from $\underline{A}^{(2)}$ and $\underline{B}^{(2)}$

$$\sigma_i^{(2)} = 2^\beta \cdot 2^{v_i^{(2)}}, \quad \tau_j^{(2)} = 2^\beta \cdot 2^{w_j^{(2)}}$$

where $v^{(2)}$ and $w^{(2)}$ are

$$v_i^{(2)} = \lceil \log_2 \max_{1\leq j\leq n} |\underline{a}_{ij}^{(2)}| \rceil, \quad w_j^{(2)} = \lceil \log_2 \max_{1\leq i\leq n} |\underline{b}_{ij}^{(2)}| \rceil.$$

From $\underline{A}^{(2)}$ and $\underline{B}^{(2)}$, $A^{(2)}$, $\underline{A}^{(3)}$, $B^{(2)}$ and $\underline{B}^{(3)}$ are obtained by

$$\begin{aligned}
A^{(2)} &= \mathrm{fl}\left((\underline{A}^{(2)} + \sigma^{(2)} \cdot e^T) - \sigma^{(2)} \cdot e^T\right), \\
\underline{A}^{(3)} &= \mathrm{fl}\left(\underline{A}^{(2)} - A^{(2)}\right), \\
B^{(2)} &= \mathrm{fl}\left((\underline{B}^{(2)} + e \cdot (\tau^{(2)})^T) - e \cdot (\tau^{(2)})^T\right), \\
\underline{B}^{(3)} &= \mathrm{fl}\left(\underline{B}^{(2)} - B^{(2)}\right).
\end{aligned}$$

From the above-mentioned discussion,

$$\begin{aligned}
\underline{A}^{(2)} &= A^{(2)} + \underline{A}^{(3)}, \qquad A = A^{(1)} + A^{(2)} + \underline{A}^{(3)}, \\
\underline{B}^{(2)} &= B^{(2)} + \underline{B}^{(3)}, \qquad B = B^{(1)} + B^{(2)} + \underline{B}^{(3)}.
\end{aligned}$$

Generally, let $\sigma^{(k)} \in \mathbb{F}^m$ and $\tau^{(k)} \in \mathbb{F}^p$ be

$$\sigma_i^{(k)} = 2^\beta \cdot 2^{v_i^{(k)}}, \quad \tau_j^{(k)} = 2^\beta \cdot 2^{w_j^{(k)}} \tag{4}$$

where $v^{(k)} \in \mathbb{F}^m$ and $w^{(k)} \in \mathbb{F}^p$ are

$$v_i^{(k)} = \lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(k)}| \rceil, \quad w_j^{(k)} = \lceil \log_2 \max_{1\le i\le n} |\underline{b}_{ij}^{(k)}| \rceil.$$

Then, $A^{(k)}$, $\underline{A}^{(k+1)}$, $B^{(k)}$ and $\underline{B}^{(k+1)}$ are obtained by

$$A^{(k)} = \mathrm{fl}\left((\underline{A}^{(k)} + \sigma^{(k)} \cdot e^T) - \sigma^{(k)} \cdot e^T\right), \tag{5}$$
$$\underline{A}^{(k+1)} = \mathrm{fl}\left(\underline{A}^{(k)} - A^{(k)}\right), \tag{6}$$
$$B^{(k)} = \mathrm{fl}\left((\underline{B}^{(k)} + e \cdot (\tau^{(k)})^T) - e \cdot (\tau^{(k)})^T\right), \tag{7}$$
$$\underline{B}^{(k+1)} = \mathrm{fl}\left(\underline{B}^{(k)} - B^{(k)}\right). \tag{8}$$

Let $A = \underline{A}^{(1)}$ and $B = \underline{B}^{(1)}$. If we compute (5), (6), (7) and (8) for $k = 1, 2, \dots$ in turn, then $n_A, n_B \in \mathbb{N}$ exist such that

$$A = \sum_{r=1}^{n_A} A^{(r)},\ B = \sum_{s=1}^{n_B} B^{(s)},\ \underline{A}^{(n_A+1)} = O_{mn},\ \underline{B}^{(n_B+1)} = O_{np} \tag{9}$$

where O_{mn} means the m-by-n zero matrix. If all $A^{(r)}$ and $B^{(s)}$ are generated by (5) and (7), then it is proved in [5] that

$$A^{(i)}B^{(j)} = \mathrm{fl}(A^{(i)}B^{(j)}), \quad 1 \le i \le n_A, \quad 1 \le j \le n_B. \tag{10}$$

Hence,

$$AB = \sum_{k=1}^{n_A n_B} C^{(k)},$$

where $C^{(1)} = \mathrm{fl}(A^{(1)}B^{(1)}), \quad \dots, C^{(n_A n_B)} = \mathrm{fl}(A^{(n_A)}B^{(n_B)})$. Therefore, a matrix multiplication can be transformed into an unevaluated sum of floating-point matrices. Let computed results for the sum of floating-point matrices by accurate summation algorithms [3, 4] be $R, S \in \mathbb{F}^{m\times p}$ respectively, then

$$|R - AB| \le 2\mathbf{u}|AB|, \quad |S - AB| \le \mathbf{u}|AB|. \tag{11}$$

(11) shows that relative accuracy is guaranteed. We define the following function

$$C = \texttt{accmul}(A, B)$$

where C is a faithfully rounded result by combining our error-free transformation and the accurate summation algorithm [3].

When we compute a matrix multiplication, a function supported in optimized BLAS tends to be used, for example, GotoBLAS2 and Intel Math Kernel Library. Because of (10), level 3 fraction for our algorithm is quite high. Therefore, the algorithm receives much benefit by such optimized BLAS in terms of computational performance. However, since we split both input matrices into sum of several floating-point matrices and the results of floating-point matrix products are saved, the algorithm requires much an amount of working space.

From the later discussion, let $A, B \in \mathbb{F}^{n\times n}$ for simple discussion. Assume that A and B are split into n_A and n_B matrices respectively. Let μ be a space for keeping an n-by-n floating-point matrix, namely, $8n^2$ bytes for binary64. The required amount of working space is as follows:

- $n_A\mu$ for splitting of A
- $n_B\mu$ for splitting of B
- $n_A n_B\mu$ for keeping the result of $A^{(i)}B^{(j)}$

Then, total amount of working space becomes

$$(n_A + n_B + n_A n_B)\mu. \tag{12}$$

In the next section, we propose a method to reduce the amount of working memory.

3. Proposed Algorithm

In this section, we use several MATLAB notations. For the simple discussion, $A, B \in \mathbb{F}^{n\times n}$ and assume that n is a multiple of a block size b, i.e. $n = bk,\ k \in \mathbb{N}$. Let block matrices be

$$\begin{aligned}
A_i &= A(b*(i-1)+1 : b*i,\ :\ ) \in \mathbb{F}^{b\times n}, \\
B_j &= B(\ :\ ,\ b*(j-1)+1 : b*j) \in \mathbb{F}^{n\times b}.
\end{aligned}$$

Then

$$A = [A_1; A_2; \dots; A_k], \quad B = [B_1, B_2, \dots, B_k].$$

Let $C := AB$ and

$$C_{ij} = C(b*(i-1)+1 : b*i, b*(j-1)+1 : b*j).$$

Then, we can obtain blockwise results by $C_{ij} = A_iB_j, 1 \le i, j \le k$. The following is an algorithm of block computations with a number of partitions k.

```
accmul_block(A, B, k)
  $d = n/k$;
  for $i = 1 : k$
    for $j = 1 : k$
      $C((i-1)d+1 : i*d, (j-1)d+1 : j*d) =$
          accmul($A((i-1)d+1 : i*d, :)*$
            $B(:, (j-1)d+1 : j*d)$);
    end
  end
end
```

For the implementation of accmul_block(A, B, k), the required working memory becomes

$$(n_A + n_B)\mu/k + n_A n_B \mu/k^2.$$

Therefore, it is shown that the working space can be reduced compared to (12).

4. Numerical Examples

In this section, we show the performance of block computations. Two matrices are generated by MATLAB built-in function

$$A = \texttt{randn}(n), \quad B = \texttt{randn}(n),$$

where the function randn(n) generates an n-by-n matrix whose elements are normally distributed with mean 0, variance 1. We tested examples on the following three environments:

- Core i7-2620M (2 cores, 16 GBytes memory)
- Xeon 5550 (totally 8 cores, 96 GBytes memory)
- Xeon 5560 (6 cores, 48 GBytes memory)

Code for block computations is compiled by Intel Parallel Studio through MATLAB external interface. For comparison, Table 1 shows computing time for pure matrix multiplication AB for various n.

Tables 2 ,3, 4 show computing time for accmul_block(A, B, k) with various n and k. $k = 1$ in tables means that accmul(A, B) is executed. Notation '-' in Table 2 means that algorithm stopped due to lack of working space. From the all tables, it is confirmed that block computations do not significantly slow down the performance of the computations, maximally 20 % slower than the original algorithm accmul.

Conclusion

We show the computational performance of blockwise computation of [5]. Numerical results show that blockwise computations do not significantly slow down the computational performance and efficiently reduce the amount of working space.

Table 1: Comparison of computing time

n	Core i7-2620M	Xeon 5550	Xeon 5650
1000	0.066	0.034	0.035
2000	0.392	0.329	0.257
4000	2.821	1.573	1.992
8000	24.21	12.91	15.64

Table 2: Comparison of computing time (Core i7-2620M)

$k \setminus n$	1000	2000	4000	8000
1	0.96	6.77	53.71	-
2	1.05	7.05	56.07	556.89
4	1.08	7.82	59.37	571.30
5	1.13	8.04	63.05	585.77

Acknowledgments

This research was supported by CREST program, Japan Science and Technology Agency (JST).

References

[1] ANSI: *IEEE Standard for Floating-Point Arithmetic*, Std 754–2008, 2008.

[2] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product, *SIAM J. Sci. Comput.*, 26, 1955–1988 (2005).

[3] S. M. Rump, T. Ogita, S. Oishi: Accurate Floating-Point Summation Part I: Faithful Rounding, *SIAM J. Sci. Comput.*, 31:1, 189-224 (2008).

[4] S. M. Rump, T. Ogita, S. Oishi: Accurate Floating-Point Summation Part II: Sign, K-fold Faithful and Rounding to Nearest, *SIAM J. Sci. Comput.*, 31:2, 1269-1302 (2008).

[5] K. Ozaki, T. Ogita, S. Oishi, S. M. Rump: Error-Free Transformation of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and its Applications, *Numerical Algorithms*, 59:1, pp. 95-118 (2012).

[6] The MPFR Library: http://www.mpfr.org/

Table 3: Comparison of computing time (Xeon 5550)

$k \setminus n$	1000	2000	4000	8000
1	0.67	4.14	32.65	272.7
2	0.67	4.29	28.29	272.7
4	0.74	4.60	29.67	265.9
5	0.77	4.81	30.72	280.0

Table 4: Comparison of computing time (Xeon 5650)

$k \setminus n$	1000	2000	4000	8000
1	0.74	5.07	36.3	339.3
2	0.76	5.14	36.8	340.3
4	0.85	5.48	37.7	345.8
5	0.86	5.65	38.4	352.4

[7] exflib - extend precision floating-point arithmetic library: http://www-an.acs.i.kyoto-u.ac.jp/ fujiwara/exflib/exflib-index.html